Quicksilver: Automatic Synthesis of Relational Queries

Edward Lu
Ras Bodik
Björn Hartmann, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-68
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-68.html

May 15, 2013

Copyright © 2013, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Quicksilver: Automatic Synthesis of Relational Queries

Edward Kuanshi Lu, Rastislav Bodik
University of California, Berkeley
Berkeley, CA 94704 USA
edward.k.lu@berkeley.edu, bodik@cs.berkeley.edu

Abstract

Relational data has become so widespread that even end-users such as secretaries and teachers frequently interact with them. However, finding the right query to retrieve the necessary data from complex databases can be very difficult for end-users. Many of these users can be seen voicing their confusion on several Excel help forums, receiving little help and waiting days for responses from experts. In this paper, we present Quicksilver, a programming-by-demonstration solution that derives queries from user inputs. It is designed to be easy and intuitive for users who are not familiar with database theory. We present Quicksilver's interface designs and synthesis algorithms. We conclude with a user study designed to evaluate Quicksilver's performance.

Index Terms

Programming-by-demonstration, Program Synthesis, Databases

I. INTRODUCTION

As relational data becomes more efficient and useful, end-users increasingly need to use it to create visualizations, data mine, design websites, and more. However, little work has been done to help the end-user transform this data to a useable state. Designing the exact query to retrieve the right contents from a database should not be a skill the end-user is expected to have. Queries can be very hard to understand, and even harder to debug, especially for large databases.

Many users experience difficulties describing the transformations they want to execute over relational data. For example, a study by Resiner evaluating SQL [7] showed that even after 14 hours of academic instruction, nonprogrammers averaged a 65% on a test asking them to write SQL queries. This study does not account for retention of knowledge, and it is reasonable to assume that most end-users would not spend that long learning SQL.

We can also see this online in Excel help forums.[1] For example, one user, a business man, has a table with over 18,000 lines of data and wishes to extract those rows that have revenues larger than a certain value. Another user, a manager for a golf society, wishes to delete unwanted columns in his records of golf scores. A third user, a teacher, has 20 worksheets, each recording different test results for his students. He would like to perform a join across all worksheets on name, into what he calls a "summary table". These users struggled with achieving their goals so they posted their question on these forums, waiting days to weeks for a response from an expert. Some of these users have yet to receive a response on the forums. They would post a short example of the tables that they have, and a few examples of what they would and would not like as the result of the transformations.

The main contribution of this paper is Quicksilver, a tool designed to become the expert for these users, providing quick responses to the user's examples. It is written as a web tool utilizing HTML5 and optional touch capabilities in an effort to be accessible anywhere. Users upload tables that they want to transform to this tool, from which Quicksilver creates a few samples. The user then drags cells from these samples to create examples of the final table they want or negative examples of what they do not want to see. If the user is not satisfied with the samples, the user can specify which rows he would like to use. Using synthesis techniques, Quicksilver can quickly respond

[1] http://www.excelforum.com, http://www.mrexcel.com/forum/excel-questions/, http://www.ozgrid.com/forum/

with its best guess at the user's intentions by providing the user its suggested query and the result of applying this query to the uploaded tables. It also highlights rows of this result table that it thinks may be incorrect. The user can examine the suggested query and the results to determine if Quicksilver's inference is correct. Should the user be satisfied, they can download the resulting table and use it in regular spreadsheet programs. Otherwise, the user can easily refine their intentions to Quicksilver by adding more examples or marking rows in the result table as incorrect. Quicksilver will quickly respond with its updated guess.

This tool is limited to the expressibility of relational algebra, as its synthesis engine is restricted to this language. This is enough to express any relational query a user may require. Transformations that involve semi-structured or non-structured tables are out of scope of this paper and have been thoroughly explored in [10].

This paper will first explain the user interaction with Quicksilver through a series of scenarios. It will then discuss the synthesis problem and the algorithms behind the tool. Additional features and related work will conclude the paper.

II. SCENARIOS

All relational queries can be described by a combination of the relational algebra primitives: projections, selections, cross products, set differences and set unions. To illustrate Quicksilver's interface and capabilities, we will show how our tool handles these primitives by walking through a few scenarios. First we describe the premise behind all the scenarios. Then we will have an overview of relational algebra. We conclude this section with 3 scenarios that describe our tool in more depth.

Professor Alice advises students every semester. One of Alice's jobs is to review each of her advisee's class schedules for the upcoming semester. Should the schedule pass her inspection, she will give the advisee their unique advising code. The student needs this code because it is required to begin signing up for next semester's classes. Alice keeps her advisees' information in a table shown in Figure 1. To facilitate this process, Alice requests students to signup for a time to meet and talk about their future class schedule. A sample of this table is shown in Figure 2.

Student	Email	Class	GPA	Adv. Code
John Doe	jd@b.edu	Senior	4.0	2315
Anne Foe	af@b.edu	Soph.	4.0	8112
Rachel Floe	rf@b.edu	Fresh.	3.3	2948
Lue Goe	lg@b.edu	Senior	3.2	

Fig. 1: A sample of the Advisee Table

Time	Name
1:00pm	John Doe
1:20pm	Anne Foe
1:30pm	Jimmy Toe

Fig. 2: A sample of the Signup Table

A. Relational Algebra

Before we begin, we will first provide a brief overview of the relational algebra. All relational queries can be described by a combination of primitives: projections, selections, cross products, set differences and set unions [4]. We will describe each of these, along with the commonly used join operation, below.

1) Projections: Projections, typically symbolized as ρ, manipulates the columns of a relational table. The operation removes columns that a user may not want to see in the table. For example, ρ_{Name} Advisees would project Name from the Advisees table. The result of this operation would be the Name column of the Advisees table.

2) Selections: Selections, typically symbolized as σ, acts like a filter across rows of a table. Selections are defined by their conditions, which we refer to as selection predicates in this paper. If the selection predicate is true on a given row, that row is outputted. Otherwise, it is ignored. For example, $\sigma_{Name==JohnDoe}$ Advisees would select any rows from Advisees where the Name column equals "John Doe".

3) Cross Products: Cross products, typically symbolized as $\times$, operates over two tables to create all possible concatenations of the two tables' rows. In psuedocode, cross products would be implemented as follows:

Algorithm 1 Cross product

```
1: procedure CROSSPRODUCT(A, B)
2:     result = empty table
3:     for row a in A do
4:         for row b in B do
5:             add concat(a,b) to result
6:     return result
```

4) Set Unions: Set unions, denoted $\cup$, operates over two tables to create one table that contains all the rows from each table without duplicates. It behaves exactly like a set union where each table is considered a set of its rows. Formally, it is defined as $A \cup B = \{x : x \in A \vee x \in B\}$.

5) Set Differences: Set differences, denoted $\setminus$, also operates over two tables. It is defined as $A \setminus B = \{x \in A : x \notin B\}$.

6) Set Operations: We refer to set unions and set differences as set operations for this paper. These are unique in that they operate over multiple tables, each of which may contain different columns from the other. In order for set operations to be valid, these operand tables must be *set compatible*. This means that for every set operation's arguments T_1, T_2, the columns of T_1 must be the same length as the columns of T_2. Additionally, the ith column of T_1 must match in type as the ith column of T_2 for all i less than the total number of columns in T_1.

7) Joins: Joins operate over two tables to create one table that contains concatenated rows from each table that satisfy a join condition. This can be expressed as a cross product followed by a selection whose selection predicate is the join condition.

We will now discuss 3 simple scenarios that describe how we handle each operation.

B. Joins

With the following scenario, we will show how our tool handles joins using a combination of cross products and selections. We will also show how we handle projections.

Alice would like to have a common table that lists each student who signed up along with their advising code. To do so, she uploads each table to Quicksilver, which directs her to the *candidate pool*, a list of tables that contain sample rows taken from the uploaded tables. From here, she can illustrate examples to Quicksilver by dragging cells from the candidate pool. She specifies a *positive example*, a tuple that she expects to see in the result, by dragging cells "1:00pm" and "John Doe" from the Signup sample into a row in the result box. She then drags John Doe's Advising Code from the Advisee sample next to this row, as shown in Figure 3.

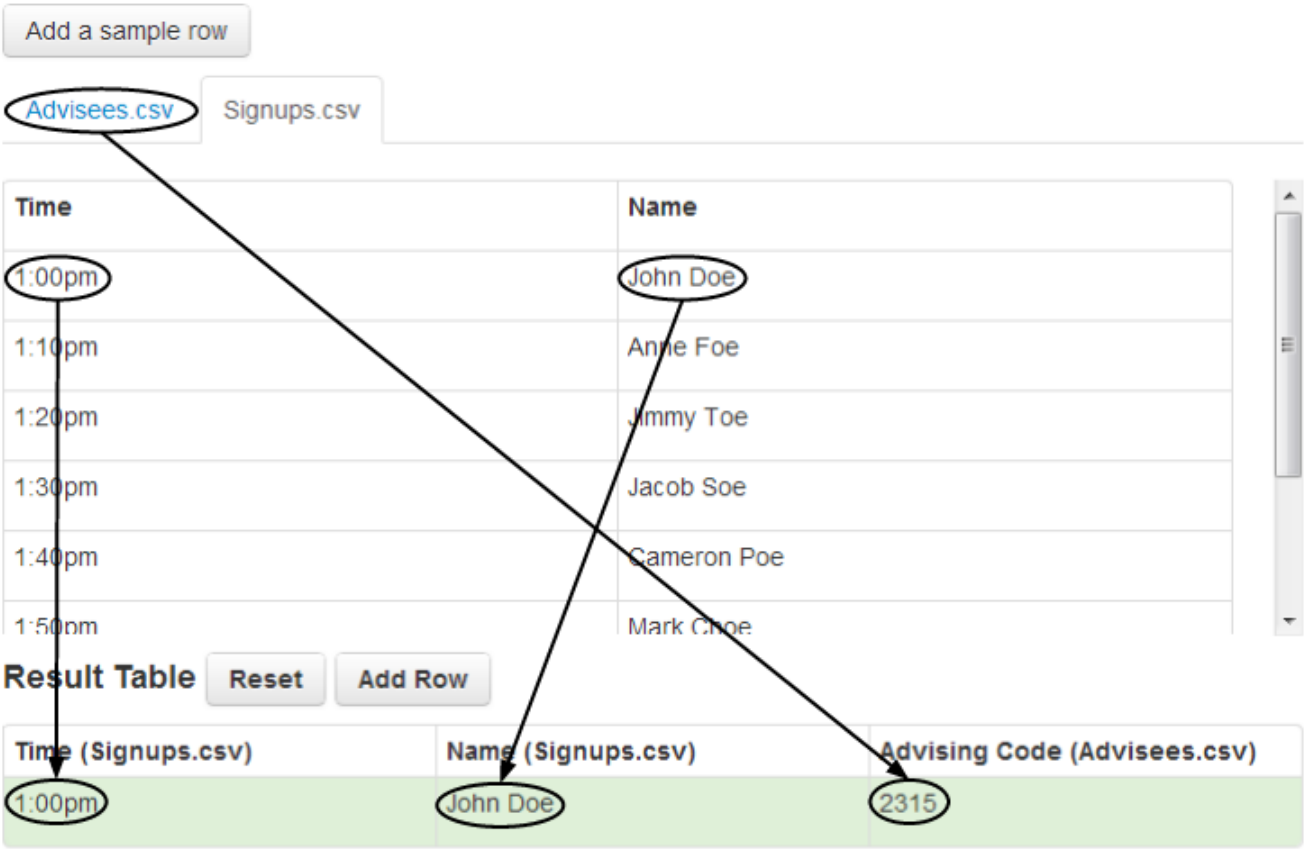

Fig. 3: Alice drags cells from the candidate pool into the result table to form a positive example (outlined by the arrows). The Advising Code cell was retrieved when Alice was on the Advisees tab. If she didn't find a good example with these rows, she could specify new sample rows to add to the candidate pool using the button "Add a sample row" above.

Once she confirms her examples, Quicksilver passes them onto its synthesis engine, producing its best guess at the query. It guesses a query within milliseconds and displays the first few rows of executing this query on her input tables, as seen in Figure 4. The query it found was:

$$\rho_c(\sigma_a(\text{Advisees} \times \text{Signups}))$$

Where c is Signups.Time, Signups.Name, Advisees.Advising Code and a is Signups.Name == Advisees.Student. In english, the query was a projection of columns c of the tuples in the cross product of the Advisees and Signups tables that satisfy the predicate a. In this case, Quicksilver's best guess was correct, so Alice is done here. However, if Quicksilver had guessed wrong, Alice could mark any of the result rows as a *negative example* (a tuple that she does not want to see in the result) by clicking the x next to the corresponding tuple. This will prompt Quicksilver to guess a query that does not produce these negative examples. She could also add another positive example to similarly cause Quicksilver to look for another query.

If Alice wanted to create this table without Quicksilver, she would most likely have to do it manually, by cutting and pasting cells into their appropriate position. This would have been time consuming, especially if these tables were large. If she had some knowledge of database programs, she may have uploaded the tables to an SQL database, figure out the correct query to execute, and put the resulting table back into a readable format. However, most end-users do not have this knowledge and working with relational database programs will be difficult. With Quicksilver, however, all Alice had to do was specify one positive example to receive what she wanted in milliseconds. One positive example here was sufficient because it expresses all required properties of the query such as the projection columns and selection predicate. The synthesis engine's ranking engine then decided on the correct query to guess

Status: Success. Query: PROJECT columns ['Time' (col 5), 'Name' (col 6), and 'Advising Code' (col 4)]] FROM SELECT where 'Student' (col 0) == 'Name' (col 6) of (SELECT where 'Student' (col 0) == 'Name' (col 6) of (a cross product of all tables)). Synthesis Time: 20.0 ms

Add a custom example

Time ×	Name ×	Advising Code ×	
1:00pm	John Doe	2315	×
1:10pm	Anne Foe	8112	×
1:20pm	Jimmy Toe	5395	×
1:30pm	Jacob Soe	1234	×
1:40pm	Cameron Poe	1245	×
1:50pm	Mark Choe	4930	×
2:00pm	Luke Boe	9283	×
2:10pm	Lucy Voe	2913	×

Fig. 4: Quicksilver's response to Alice's join examples. All of the advisees who signed up have their advising code listed next to them. The query found is shown above, and the results of the query follow it.

that satisfied these properties.

C. Selection

Alice notices that her Advisee table is incomplete. In particular, students like Lue Goe do not have an advising code (Figure 1). She would like a table of all the students who do not have advising codes, so she can send the names to administration to ask for the codes. Alice goes to the candidate pool and creates one example from the cell with Lue Goe's name. She sends this example to Quicksilver, which synthesizes the following query:

$$\rho_{Name}(\text{Advisees})$$

The results of this query is shown in Figure 5. She notices that this cannot be the right query, as she knows that John Doe does in fact have an advising code. So she refines the query by clicking the $\times$ button next to the John Doe row. Quicksilver resynthesizes, and finds the following query:

$$\rho_{Name}(\sigma_a \text{Advisees})$$

Where a is Advising Code == null. The results is shown in Figure 6. This is the correct query, so Alice is satisfied.

D. Set difference

In this next scenario, Alice notices that Rachel Floe, an advisee, hasn't signed up yet (Rachel Floe is in Figure 1, but not in Figure 2). She wonders if anyone else hasn't signed up yet so she can email them a reminder. Alice returns to the candidate pool and creates two examples, as shown in Figure 7. The first example is positive, showing that she wants to see the name Rachel Floe. The second example is negative, indicating that she does not want to see John Doe, a student who has signed up. Notice that the negative example is colored red. Alice can toggle between positive and negative examples by clicking the tuple. Quicksilver sends visual feedback in the form of color: green being positive and red being negative. Once Alice confirms her examples, Quicksilver synthesizes the following query:

$$\rho_a(\text{Advisees}) - \rho_b(\text{Signups})$$

Status: Success. Query: PROJECT columns ['Student' (col 0),] FROM a cross product of all tables. Synthesis Time: 20.0 ms

Add a custom example

Student ×	
Lue Goe	×
John Doe	×
Anne Foe	×
Jimmy Toe	×
Jacob Soe	×
Jennifer Moe	×
Jeff Yoe	×
Matthew Loe	×
Cameron Poe	×
Mark Choe	×
Luke Boe	×
Lucy Voe	×
Patrick Noe	×
Rachel Floe	×

Fig. 5: Alice tries to find the names of advisees without advising codes.

Where columns a and b are Advisees.Student and Signups.Name, respectively. This is the correct query. The query is executed on the inputs (larger than what is shown in Figures 1 and 2), and Quicksilver shows Alice the results, shown in Figure 8.

E. Unions

Unions can be similarly constructed through the use of multiple examples originating from different columns. For example, if Alice wanted to see the names of all her advisees and all the students who signed up, she could provide Quicksilver with any name from Advisee.Student and any name from Signup.Name. This would be sufficient for the tool to produce a union of Advisee.Student and Signup.Name columns with the following query:

$$\rho_a(\text{Advisees}) + \rho_b(\text{Signups})$$

We will next discuss our synthesizer. We will first describe its inputs as a language of demonstrations, then its output as a language of queries, and finally the algorithms that connect the two languages.

III. Demonstrations

Quicksilver's synthesizer retrieves inputs as user demonstrations. These demonstrations can be represented as a series of *drag actions* and *mark actions*.

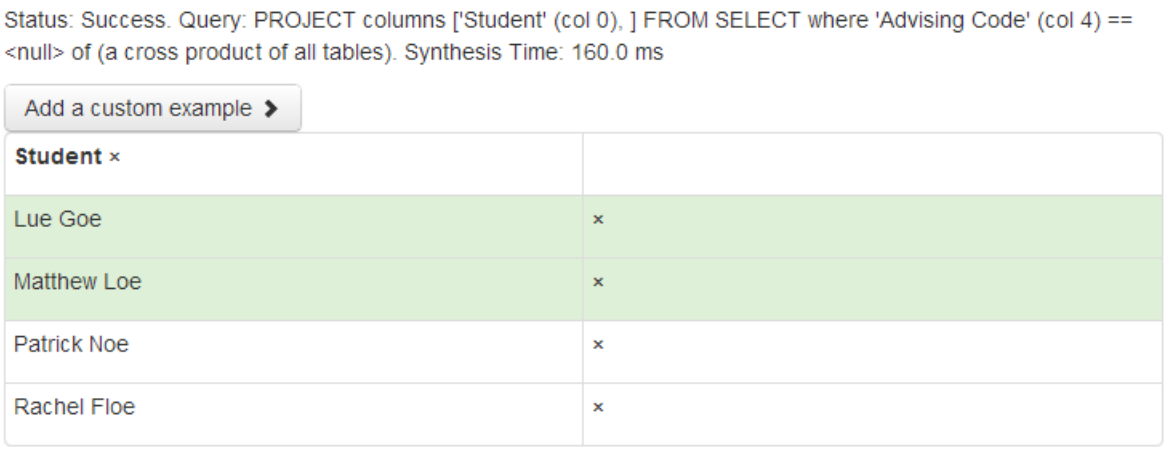

Fig. 6: The results after Alice refines her original selection examples.

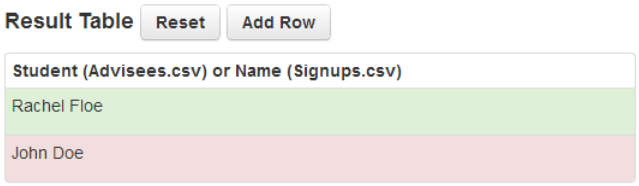

Fig. 7: Alice tries to find the names of advisees who have yet to sign up with these examples.

A. *Drag Actions*

A drag action is when the user drags a cell from the candidate pool into the *result table*, a table where the user can specify positive or negative examples. We represent this action as `drag(t,c,r,ec,er)`, where `(t,c,r)` represent the table, column, and row of the cell's original location (i.e. the one in the uploaded file) and `(ec, er)` represent the column and row of the cell's new location (i.e. the one in the result table). For example, the action involving the "John Doe" cell in Figure 3 is represented as `drag(Signups,1,0,1,0)`. Drag actions retain information about the destination location of the cell in order to construct the examples. These actions contain information about the source location in order to aid synthesis by limiting the search space. In particular, the location can tell us which projection columns to use and hint at whether or not we need to use set operations. We will explain this in more depth when we discuss *projections sets* in the Synthesis section.

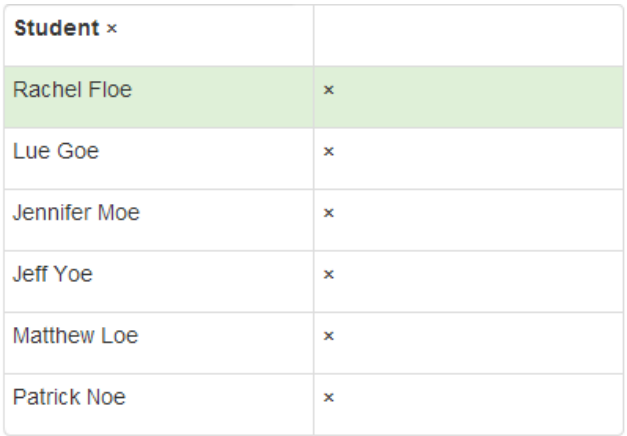

Fig. 8: Quicksilver's response to Alice's set difference query.

$$
\begin{array}{rcl}
D & := & D, d \mid D, m \mid d \mid m \\
d & := & \texttt{drag(t,c,r,c,r)} \\
m & := & \texttt{mark(r,}e\texttt{)} \\
e & := & \text{positive} \mid \text{negative} \\
t & := & \text{table} \\
c & := & \text{column} \\
r & := & \text{row}
\end{array}
$$

Fig. 9: The demonstrations grammar

$$
\begin{array}{rcl}
Q & := & \texttt{unique}(D) \mid D \\
D & := & \texttt{diff}(D, D) \mid E \\
E & := & \texttt{project}(S) \\
S & := & \texttt{select}(i, P) \\
P & := & P, (C, A) \mid (C, A) \mid null \\
A & := & A, c \mid c \\
C & := & C \wedge C \mid c = d \mid c \neq d \mid c > d \mid c < d \\
d & := & c \mid v \\
c & := & \text{any column in } I \\
i & := & \text{the cross product of } I \\
v & := & \text{any cell value in } I
\end{array}
$$

Fig. 10: A grammar of synthesized queries

B. Mark Actions

A mark action is when the user defines an example row as either a positive or a negative example. The user can do this by clicking the row, toggling its color from green (positive) to red (negative). As mentioned earlier, a positive example is an example that must be present in the result and a negative example is one that must not be present in the result. Mark actions are represented as `mark(r,`e`)`, where `r` is the row number of the result table and e is either positive or negative.

Our demonstrations grammar can be seen in Figure 9. We designed the grammar to be simple so that it would lend to a simpler user experience.

IV. Queries

Quicksilver's output consists of the relational algebra primitives: projections, selections, cross products, set unions, and set differences. As mentioned before, these operators are supported so that we can output any relational algebra query the user may request. The grammar of output queries can be seen in Figure 10.

We enforce an order on the grammar of these queries in order to limit the search space of our synthesizer. We order them as, from first to last: cross product, selection, projection, set union, and set difference. We will discuss how each operation is defined and the completeness this grammar in this section.

Each operator is defined in two different formats: (1) as functions, and (2) as a set of constraints in SMT format. (1) is needed to interpret the results of our synthesis. (2) is necessary for our synthesis engine. We will describe these operations in format (1) for this section, and in format (2) in our algorithms section.

A. Cross products

The cross product operation is defined in only one instance, a cross product of the input relational tables, I. I is constructed as follows: for each `drag(t,c,r,ec,er)`, we extract the row `r` from table `t` add it to an initially empty representation of `t` in I. All queries we generate are functions of this cross product of I.

B. Selections

Selections are defined as `select`(i, P), where i is the cross product of I, and P is a list of pairs of selection predicates and *column lists*. A column list is defined as a list of columns, which will be used for projections. The pairing is used for set unions, which will be explained later. For each pair $p \in P$, the `select` operator evaluates p's selection predicate on every row r in i. If p evaluates to true, `select` will add (r, p's column list) to a list. This list is what `select` will return upon termination. In pseudocode:

Algorithm 2 Select as a function

```
1: procedure SELECT(i, P)
2:     result = empty list
3:     for r in i do
4:         for (predicate, column list) in P do
5:             if predicate(r) then
6:                 add (r, column list) to result
7:     return result
```

C. Projections

Projections are defined as functions that are executed on a selection's results. Given a list of (row r, column list l) tuples L, `project` will, for each tuple, create a new row consisting of the columns of r that exist in l, in the order they appear in l. In pseudocode:

Algorithm 3 Project as a function

```
1: procedure PROJECT(L)
2:     result = empty list
3:     for r, l in L do
4:         row = empty list
5:         for column in l do
6:             add r[column] to row
7:         add row to result
8:     return result
```

D. Set unions

Set unions are defined inside selections as selection predicate, column list pairs. We can do this because each operand of a set union, u_i, can only differ from any other operand, u_j, in projections and selections. They cannot differ in cross products, because all other primitives operate over the same cross product of I. They cannot differ in set difference, because of the order of operations we impose. Because of this, we can associate each u_i with a (selection predicate, column list) pair. If the selection of u_i does not filter a row r out, then the selection predicate will return true. r will then be projected as dictated by u_i using the pair's column list. This will occur in the `select` and `project` stages.

E. Set differences

Set differences are defined as functions of two lists of rows, l_1 and l_2. For each row in l_1, we check that it does not belong in l_2. If it does not, we add it to the result list. Otherwise we ignore it. In pseudocode:

Algorithm 4 Set difference as a function

```
1:  procedure DIFF(l_1, l_2)
2:      result = empty list
3:      for r_1 in l_1 do
4:          match = false
5:          for r_2 in l_2 do
6:              if r_1 == r_2 then
7:                  match = true
8:                  break
9:          if match then
10:             add r_1 to result
11:     return result
```

F. Set compatibility

We do not reason over column types. Therefore we do not make any guarantees about set compatibility. We assume two rows are set compatible as long as their lengths are equal.

G. Uniqueness

If set operations are involved, we make sure that there are no duplicate rows. We do this by executing `unique` on the final list of rows. We can detect if set operations are involved by checking if P in `select` is greater in length than 1 (implies set unions) and if `diff` was executed.

H. Completeness

There are two aspects of this grammar that limit the queries we can create: (1) limiting cross products to cross products of I, and (2) limiting the order of our operations as cross product, selection, projection, set union, and set difference.

Before we continue, we define *query equivalence* as the following: two queries a and b are strictly equivalent if the result of executing either of them on any input is indistinguishable. a and b are λ-equivalent if the result of executing either of them on the set of input tables λ is indistinguishable. a and b are partial-equivalent if they are λ-equivalent for some λ but not strictly equivalent.

We argue here that limitations (1) and (2) only prevent us from creating queries that are either not useful or queries that have strictly equivalent queries that we can create.

Let us first discuss limitation (1). We first discuss the input to the cross product, I. For each `drag(t,c,r,ec,er)`, we extract the row `r` from table `t` add it to an initially empty representation of `t` in I. This implies that we do not consider cross products with uploaded tables whose cells are never used in examples, since these tables would never be in I. For example, if Alice only dragged cells from the Signups table, Quicksilver will never consider data from the Advisees table. We believe it is reasonable to assume that we do not need tables that are not mentioned in the examples.

Another implication of (1) is that we do not consider cross products of a superset of I. This is reasonable we have no reason to use inputs outside of what the user provides. Also, we do not consider cross products of elements of I that occur more than once (e.g. Advisees $\times$ Advisees $\times$ Signups). We assume that the user does not want queries like these, because they always produce duplicate information.

Similarly, we do not consider cross products over a strict subset of I. For example, if Alice dragged cells from both the Signups and Advisees table, Quicksilver will never consider queries that execute over data from only one of these tables. To explain why this is acceptable, we will discuss two classes of queries: queries without set

operations, and queries with set operations. Queries without set operations will never need to operate over data from a strict subset of I. This is because I contains all of the user's examples. If we construct a query that operates over a strict subset of I, then we are missing data from some of the user's examples. Thus we do not satisfy the user's examples and this query is incorrect. Queries with set operations may legitimately involve data over a cross product of a strict subset of I. For example, consider our previous scenario with Alice when she wanted to see a table with both her Advisees' names and Signups' names in one column. The query is repeated here:

$$\rho_{Student}(\text{Advisees}) + \rho_{Name}(\text{Signups})$$

In this case, I consists of rows from both the Advisees and Signups tables. Thus R would be a cross product of these tables. We want a union of two subqueries: $s_0 = \rho_{Student}(\text{Advisees})$ and $s_1 = \rho_{Name}(\text{Signups})$. Each of these subqueries operates over only one table, which we consider a cross product of a strict subset of I (here we refer to a single table as a cross product of one table for generality). This means we need to pass as arguments to these queries rows that do not belong in a cross product of I. However, we can show that this is not a problem with Theorem 1.

Theorem 1 Let T be any relational algebraic query whose domain consists of a set of cross products of strict subsets of I and that the union of these subsets equals I. Let Ω be the set of queries whose domain is R. There exists a query $\omega \in \Omega$ such that ω is *strictly equivalent* to T.

Proof Every row that exists in T (i.e. passes the selection predicates and does not exist in the right hand side of a set difference) is a subset of at least one row in R. This is true because R is a cross product of I whereas T's domain contains cross products of strict subsets of I. Because of this fact, we can see that if T and ω have the same row existence criteria (i.e. same selection predicates and set operations), the rows that exist in T will be a subset of the rows that exist in ω. It is a subset because many rows in the cross product of I may be supersets of a row in T. If T and ω have the same projections, then the rows that T produce will be the same as ω minus some duplicate rows in ω's results.

T's domain consists of a set of cross products of strict subsets of I and the union of these subsets is I. This implies that T can be factored into at least two subqueries, s_i, such that each subquery operates over a cross product of a unique subset of I, i_i. Because each subquery operates over a different i_i, they have different column lists, so we know that the operation between these subqueries is a set operation (only set operations can have operands of differing column lists). One feature of set operations is that all duplicate rows are removed. Thus the result of T must have no duplicates. If we choose a ω such that its query structure is exactly the same as T, we can see that all of ω's potential duplicate rows would have been removed by these set operations. Thus T is strictly equivalent to ω.

The ordering limitation (2), does not limit our expressiveness. We can see this by the fact that any query q_0 whose domain is $\subseteq I$ produces results that can be replicated with a query q_1 whose domain is R, the cross product of I, with the same reasoning as Theorem 1. Any selection can be pushed to immediately after the cross product, because every column that could be used in the selection predicate exists in R. Projections can be pushed to the end of these selections, because these operations do not affect any other operation except for selections. If a projection happens before selection, it may potentially remove data that a selection may need for a predicate, thus projections must happen after a selection in order to not lose expressiveness. Set unions and set differences occur at the end. This is acceptable because a projection that happens after set operations can happen before them with no side effect. A selection that happens after set operations can be factored inside each operand of these set operations. Finally, a cross product of set operation equations can be equated to set operations of R. We can see this because any set operation equation cannot produce tuples outside of R, so we can trivially select and project only those tuples that belong in this cross product of set operation equations.

V. ALGORITHMS

Our main algorithm's goal is as follows: given user specified positive examples P, negative examples N, and input tables I, find a query Q such that executing Q on I will produce a set of rows, $Q(I)$, such that $Q(I)$ is either

equal or a superset of P, and there does not exist a negative example n in N such that n is in $Q(I)$.

We divide this into two problems. We first find a query Q such that P exists in $Q(I)$. This is our *synthesis* step. We then check if any $n \in N$ exist in $Q(I)$. This is our *verification* step. If no such n is found, we terminate and return Q as our best guess at the user's intentions. Otherwise, we return to the synthesis step to find a different Q. We loop until we find a Q that does not contain N, or until we cannot find anymore queries. We discuss these steps in more detail below.

A. Synthesis

Quicksilver's synthesis algorithm is designed to solve the following problem:

$$\exists Q\Big(\forall p \in P\Big(\exists r \in R \mid Q(r) == p\Big)\Big) \tag{1}$$

Where P is the set of positive examples provided by the user, Q is the query we are solving for, and R is the cross product of the input relational tables, I. The query Q is constructed as described in our discussion of our query language. (However, Q does not contain set differences. This is implemented in the synthesis-verification loop and will be discussed later.)

To state the problem in other words, the synthesizer first finds the cross product of I. For each positive example, it will find a row r from this cross product such that $Q(r)$ is equivalent to said example.

This algorithm is encoded as SMT constraints. We will discuss these constraints in the order of Q's construction as dictated by the query grammar in Figure 10.

1) Cross products: The cross product of I, R, is the input to our synthesis algorithm. We first encode each table in I, T_i, as a bitvector. Then, for each positive example p, we encode a row r_p as follows:

Algorithm 5 Encoding r_p

```
start = 0
end = 0
for T in I do
    end += length of the columns of T
    assert r_p from indices start to end equals a row from T
    start += length of the columns of T
```

Since r_p is a concatenation of a row from each table in I, r_p is in the cross product of I.

2) Selections and Set Unions: Selections and set unions are defined in Q as selection predicates and column lists. Before we describe how we encode these, we first introduce the *selection atom*. A selection atom is defined as an operation over c_1 and c_2, or c_1 and v, where c_i is any column in I and v is any value in I. We encode selection atoms in four values: sel_ite, sel_c1, sel_c2, and sel_v. If sel_ite is 0, the selection atom is of the form $c_1 == c_2$, otherwise it is of the form $c_1 == v$. To support more operations (such as inequalities), we simply increase the range of values sel_ite can take and have it decide which operator to implement. However, we do not show this in this section for the sake of brevity. sel_c1 encodes c_1, sel_c2 encodes c_2, and sel_v encodes v. We fix the number of selection atoms in our ranking phase (which happens before synthesis), which we describe later. Next, we introduce the *selection aggregate*, α. This is defined as follows: for a row r, we assert that if the selection atom a_i is true on r, $\alpha[i] = 1$, otherwise $\alpha[i] = 0$.

Now we describe how we encode selection predicates and column lists. Column lists are encoded as follows: for each example $p \in P$, we extract p's column list and encode it as a bitvector, l_i. Each selection predicate, s, is

encoded as three indices, i_0^s, i_1^s, and i_l^s. i_l^s refers to which column list s is connected to. The other indices index the selection aggregate, α. A selection predicate with indices i_0^s, i_1^s is the conjunction of all selection atoms that influence $\alpha[i_0^s : i_1^s]$. In other words, if there is a 0 in $\alpha[i_0^s : i_1^s]$, selection predicate s is false. Otherwise it is true. Like with selection atoms, we fix the number of predicates we will generate before the synthesis in our ranking phase. The pseudocode for our encoding is shown below.

Algorithm 6 Encoding selection and set unions

```
1:  procedure ENCODESELECTIONANDUNIONS(columnLists, numPredicates, numAtoms, r, columns, values)
2:      α = empty array                                              ▷ the selection aggregate
3:      for i from 0 to length(columnLists) do
4:          assert l_i == bitvector(columnLists[i])
5:      for i from 0 to numAtoms do
6:          assert sel_ite_i == 0 or 1
7:          assert sel_c1_i == a column in columns
8:          assert sel_c2_i == a column in columns
9:          assert sel_v_i == a value in values
10:         if sel_ite_i == 0 then
11:             assert (r[sel_c1_i] == r[sel_c2_i] and α[i] == 1) or (α[i] == 0)
12:         else
13:             assert (r[sel_c1_i] == sel_v_i and α[i] == 1) or (α[i] == 0)
14:     assert previous_0 = 0
15:     for s from 0 to numPredicates do
16:         assert i_0^s == previous_s
17:         assert i_0^s < i_1^s < length(α)
18:         assert 0 <= i_l^s < length(columnLists)
19:         assert previous_{s+1} == i_1^s                           ▷ no selection predicate indices overlap
```

3) Projections: Finally we project each r_p and assert that it equals p. We first have to make sure that the selection predicate that is associated with p's column list is true. Thus we assert that one of the i_l^s is equal to the index that points to p's column list, and that the associated predicate must be true. In pseudocode:

Algorithm 7 Encoding projection

```
1:  procedure ENCODEPROJECTION(p, r_p, α, selectionPredicates, p's column list index)
2:      assert one of i_l^s in selectionPredicates == p's column list index
3:      for (i_0^s, i_1^s, i_l^s) in selectionPredicates do
4:          if i_l^s == p's column list index then
5:              for j from i_0^s to i_1^s do
6:                  assert α[j] == 1
7:      for column in p do
8:          assert r_p[column] == p[column]
```

B. Verification

To support negative examples, we introduce a CEGIS loop (synthesis-verification loop) into our synthesis. The loop first synthesizes as described in the previous section. The resulting query is passed onto our verifier, which solves the following:

$$\exists n \in N \Big(\exists r \in R \mid Q(r) == n \Big) \tag{2}$$

Where N is the set of negative examples. Should the verifier be unsatisfied, we know that Q does not produce any negative example and the CEGIS loop exits. If the verifier is satisfied, we know that Q is not the query the user desires. We then take the values that define Q's behavior and assert that the next synthesis cannot assign

Q to these values. These restrictions will continue to expand until either the verifier or the synthesizer is unsatisfied.

During one set of user interactions, Quicksilver will save the set of restrictions for future use. If the user defines a new negative example, these restrictions are guaranteed to be a subset of the restrictions that will be found in the new CEGIS loop. Thus by saving these restrictions now, we save time for future user interactions.

We require a CEGIS loop because it is faster to find a counterexample through our verifier than to exhaustively search every row created by Q to make sure that none of these rows are in N.

C. Set difference

If we never find a Q that satisfies the synthesizer, this implies that either (1) the synthesis step cannot find a Q such that it satisfies all postive examples, or that (2) the synthesis step cannot find a Q such that the verifier would be unsatisfied (i.e. $Q(I)$ contains a $n \in N$). The case of both happening would be considered case (1), because the synthesis step would never have completed in time for the verifier to be unsatisfied. In case (1), we terminate and suggest possible errors in the user's inputs (see section on additional features below). In case (2), we consider set differences in Q.

In case (2), there exists some set of negative examples in N, G, such that our CEGIS loop cannot find a satisfying query Q that does not contain G. Thus we introduce a new subquery, Q^-, to be part of a new query $Q = \texttt{diff}(Q^+, Q^-)$, where `diff` is the set difference operator described in our query language. We run our CEGIS loop to create both Q^+ and Q^-. Each subquery is created using our inputs I and a subset of the examples that were provided for Q. Q^+ is given P and G as its positive examples, because we know we can't synthesize a query Q such that $Q(I)$ contains P but does not contain G. We also give Q^+ $N \setminus G$ as its negative examples, as we know that we can synthesize a query $Q(I)$ that contains P and does not contain $N \setminus G$. For Q^-, we give it G as its positive examples, because we want Q^- to remove G from the set of rows $Q^+(I)$ in order for us to achieve a Q that satisfies all of P and N. We also give Q^- P as its negative examples, because we do not want Q^- to remove any positives examples from $Q^+(I)$, as it would ultimately remove these examples from Q. Note that this part of the algorithm is recursive, so we can have any nesting of set differences and are not limited to one set difference. If we find Q^- and Q^+, we have found a Q that satisfies the user examples and we terminate.

VI. ADDITIONAL FEATURES

A. Ranking

Quicksilver ranks candidate queries preemptively. We use a heuristic similar to that of Occam's Razor: the simplest query is most likely the one the user wants. We rank queries by restricting the kinds and number of instances of operations Q can consist of. Should the synthesizer be unsatisfied, we relax the restriction a little and repeat. The first query that satisfies synthesis would be the simplest query we could find and our best guess at the user's intentions. We rank complexity using the following hierarchy: selections over columns $<$ selections over values $<$ set operations.

B. Questionable Results

Quicksilver highlights result tuples that may need user inspection. These tuples are "questionable", or the tuples that are most likely wrong if our best guess is incorrect. We mark questionable tuples by synthesizing two queries instead of one. The second query would be our next best guess. We then mark the result tuples of our best guess that do not exist in our next best guess as questionable.

C. Input Noise

If the user provides erroneous examples, Quicksilver may not find the right query. It may even give up once the query complexity reaches a high enough threshold. In this case, we want to be able to suggest to the user examples

that may have been inputted incorrectly. We do this as follows:

Let E be the list of all user specified examples. Let q_i be the query produced by Quicksilver on $E - e_i$ for each $e_i \in E$. Let q_j be the query with the least complexity. This query is most likely to be correct. Thus, e_j is most likely to be the incorrect user example.

VII. IMPLEMENTATION

Quicksilver's interface is built on Python and Javascript. Uploaded files are parsed for tabular data and a few rows are chosen from each table to present to the user as the candidate pool. The candidate pool is a tab-separated list of uploaded tables. Each cell in the table can be dragged and dropped into a "Result Table" by mouse or by touchscreen. The Result Table consists of a user-specified number of rows. This is the destination of `drag` actions. When the user clicks on these rows, the user creates a `mark` action. Each click will cycle the mark type of the row.When the user confirms their examples, the Result Table's rows are sent to the synthesizer.

The synthesizer and verifier run on Microsoft Research's Z3 SMT solver and is encoded entirely in bitvector logic for speed. The synthesis results are translated into a relational query and is applied to the uploaded tables until a user-specified number of result tuples are produced. These are then shown to the user. The user can then refine the query by marking tuples as incorrect (using `mark` actions), or by adding new examples. The user may also remove incorrect examples that may contradict the correct query. This interaction continues until the user is satisfied.

When executing a query with set differences on the uploaded tables, we first execute Q^-. We then execute Q^+ and filter any results from Q^+ that belong in Q^-. Also, when the query involves set operations, we make sure to filter out any duplicate tuples.

A. Translators

Another part of Quicksilver's implementation are its *translators*. We define translators as functions that translate a synthesized query into another form. We created two translators for Quicksilver: an optimizer and an english translator.

The ordering of operations for our synthesized query does not lend to fast query execution. For instance, doing cross products before projections would result in the creation of unnecessary data. Thus we implemented an optimizer that implements a few simple heuristics, such as pushing projections and selections as early as possible, and pushing expensive operations such as cross products as late as possible.

Users may benefit from reading the query that is synthesized. However, the synthesized query is simply a series of bitvector values and indicies. Thus we implemented an english translator that converts these values into a more human readable format.

VIII. EVALUATION

We conducted a user study to evaluate how users interact with our system. We wanted to learn if users, especially ones without programming knowledge, would find our tool easier to use than their own methods. We describe the methodology, measures, demographics, and results in this section.

A. Hypothesis

We hypothesize that, given the choice of using our tool or using anything else that comes to the user's mind, the user would prefer using our tool to perform table transformations. We also hypothesize that most users will not be able to efficiently (i.e. not manually copying and pasting cells) complete common table transformations without using our tool.

B. Demographics

We surveyed 12 people from various backgrounds. We asked them the following questions and graph our demographics based on the answers to these questions.

1) How familiar are you with spreadsheet programs (scale of 1-10)?
2) How familiar are you with relational databases (scale of 1-10)?
3) How many years of computer science experience have you had?

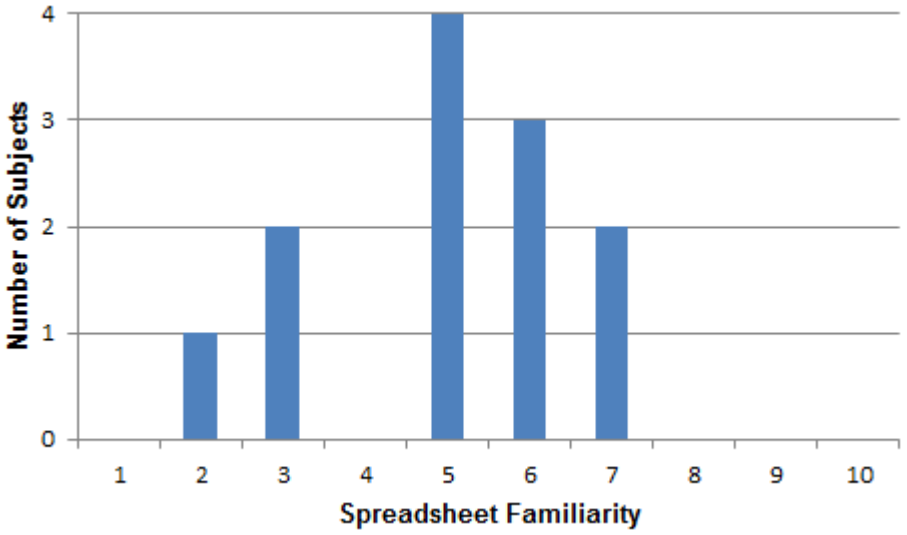

Fig. 11: Our participants' familiarity with spreadsheets.

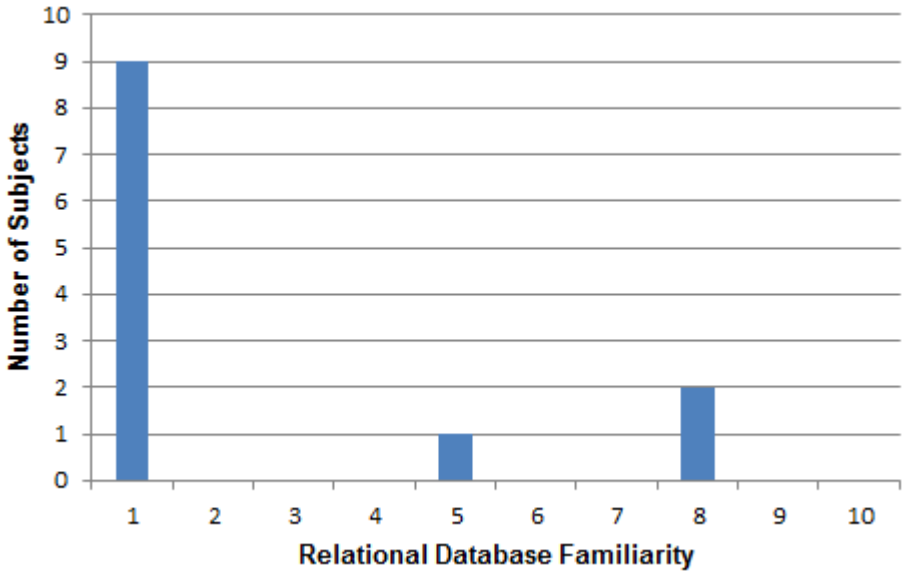

Fig. 12: Our participants' familiarity with relational databases.

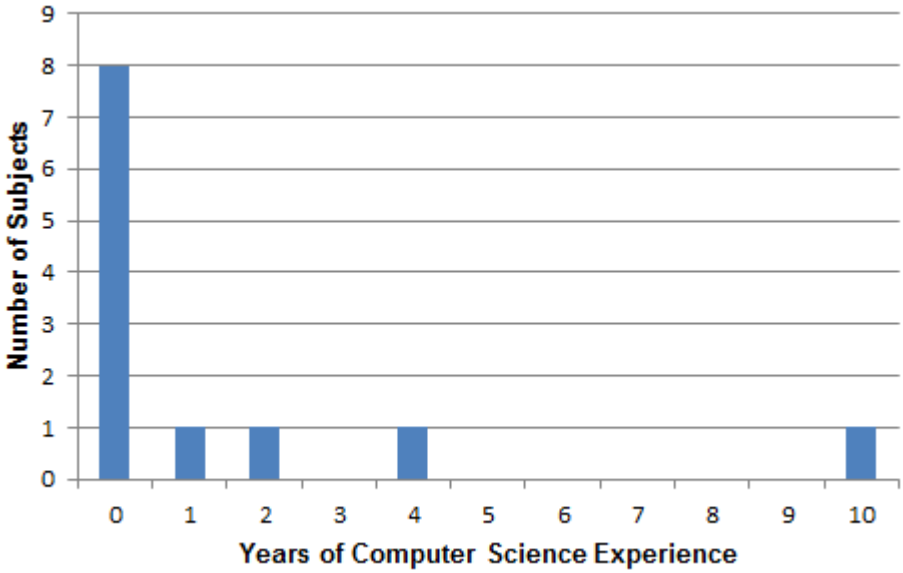

Fig. 13: Our participants' experience with computer science.

Most of our participants were somewhat familiar with spreadsheets. To put the scale in perspective, we said that anyone with a score of 6 or higher knows some Excel-like functions. A score of 10 would indicate that the subject knew most of the functions available to spreadsheet programs. A score of 1 would indicate that the subject never uses spreadsheets.

A great majority of our participants did not have any experience with relational databases (including SQL). One knew SQL, and two others knew high level relational database theory.

Most of our participants fit our target audience, as they had little to no computer science experience. We had 3 participants with more than one year of computer science experience to see if they would work as well with Quicksilver as the others.

C. Methods

We performed a within-subjects study with two conditions. In the experimental condition, participants used Quicksilver to perform two data transformation tasks after a 5 minute tutorial on the tool. In the control condition, participants performed the same transformation tasks using any method they preferred. We provided the participants the data in a browser-based spreadsheet application (Google Spreadsheets), but they were allowed to download the data and use it on any other program. Such spreadsheets are the status quo tool that Quicksilver's target audience uses today. Any questions that do not refer to the tool were answered. Questions that did refer to the tool were not answered after the tutorial. Ordering of conditions was randomized to account for learning effects.

1) Tables: The data transformation tasks used two data tables. The first is a 1000 tuple long table of advisees, similar in format as the one shown in Figure 1. It has fields for Name, GPA, Email, and Advising Code. We made this table 1000 tuples long to mimic the difficulties of operating with large data sets. We will refer to this table as the Advisee table. The second table is a 650 tuple long table of signups, similar in format as the one shown in Figure 2. We will refer to this table as the Signup table. It has fields for Time, and Student. All students mentioned in the Signup table is present in the Advisee table, but not vice versa.

2) Task A: Task A involved asking the subject to produce a table similar to the one described in the Joins section of our Scenarios discussion. In other words, we asked the subject to create a table that lists everyone who signed up, next to their signup time, next to their advising code for only those people who signed up. The query

we requested is shown below.

$$\rho_a(\sigma_b(\text{Advisee} \times \text{Signup}))$$

Where a is Name, Time, and Advising Code and b is Advisee.Name == Signup.Student. We gave the subject a 20 minute time limit. They were allowed to give up at any time.

3) Task B: Task B involved asking the participant to create a table that lists every advisee who did not sign up, with the same time limit. The query we requested is shown below.

$$\rho_{Name}(\text{Advisees}) \setminus \rho_{Student}(\text{Signups})$$

They were also allowed to give up at any time during this task.

4) Measures: Our main quantitative measures are task success, task completion time, and the number of iterations it took for the subject to solve each task (only applicable to the experimental condition). We also elicited user preferences. We define an iteration as one communication from the subject to the synthesis engine. In other words, the number of iterations is equal to the number of inferred queries.

5) Questions: To gain a more nuanced understanding of the strengths and shortcomings of each tool, we also asked a set of qualitative questions listed below.

1) (If the user did not complete Task B) If you had more time, what would you have done for Task B?
2) Did you prefer using Quicksilver or the method you came up with in Task B?

D. Results

We first present the graphs of our recorded measures. We will then finish this section with an analysis of these graphs and the responses to our qualitative questions.

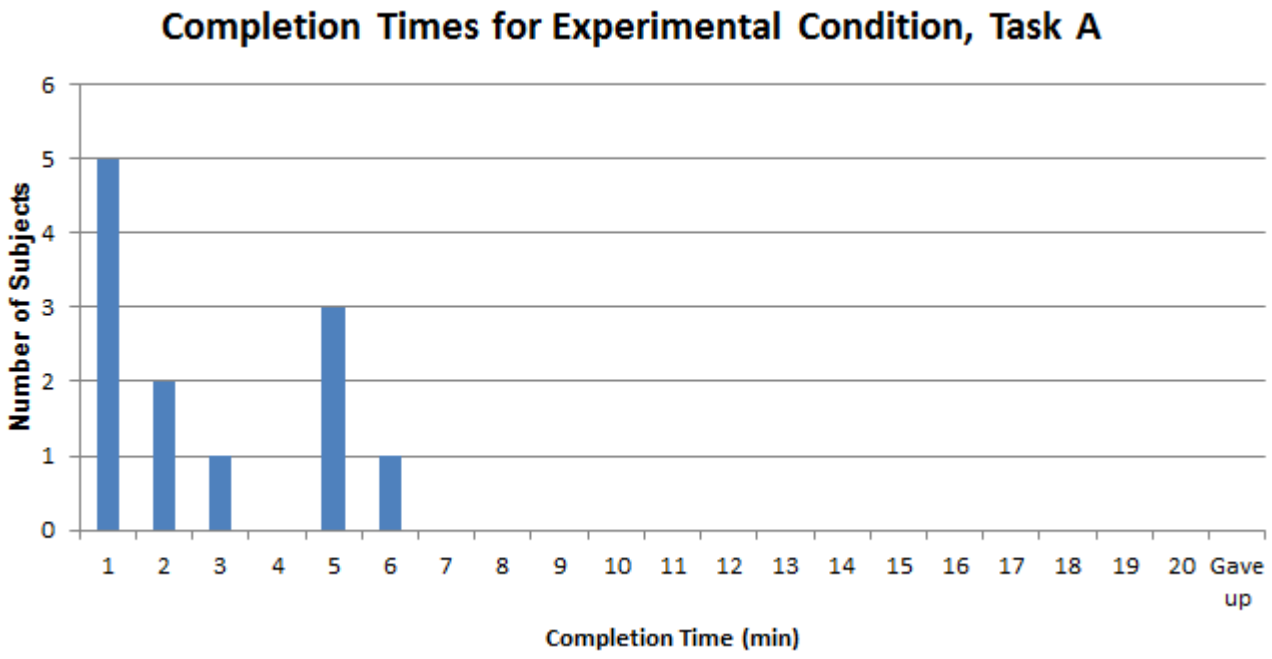

Fig. 14: Our participants' completion times for experimental condition task A.

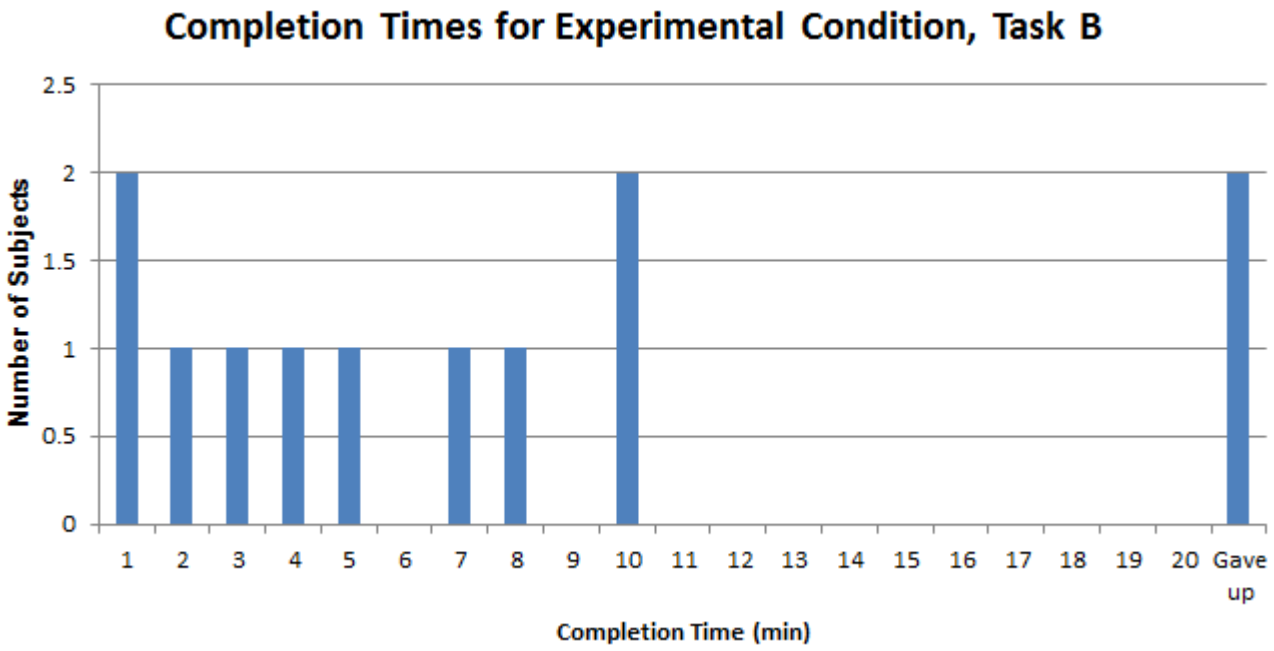

Fig. 15: Our participants' completion times for experimental condition task B.

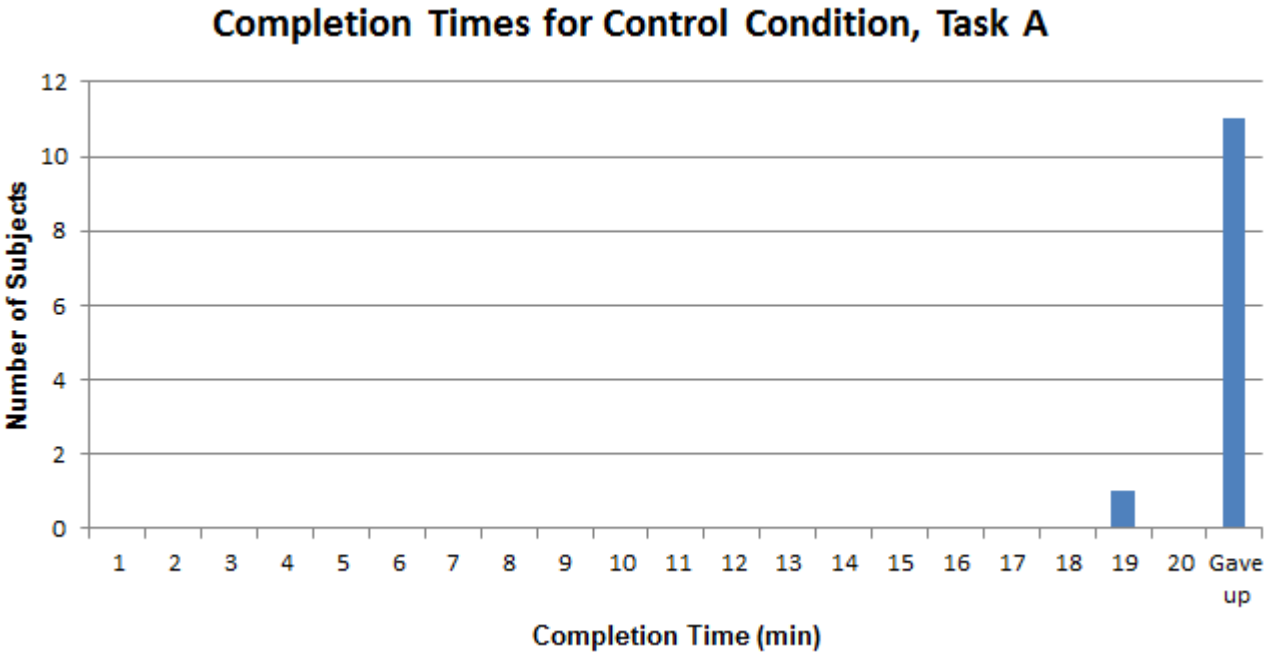

Fig. 16: Our participants' completion times for control condition task A.

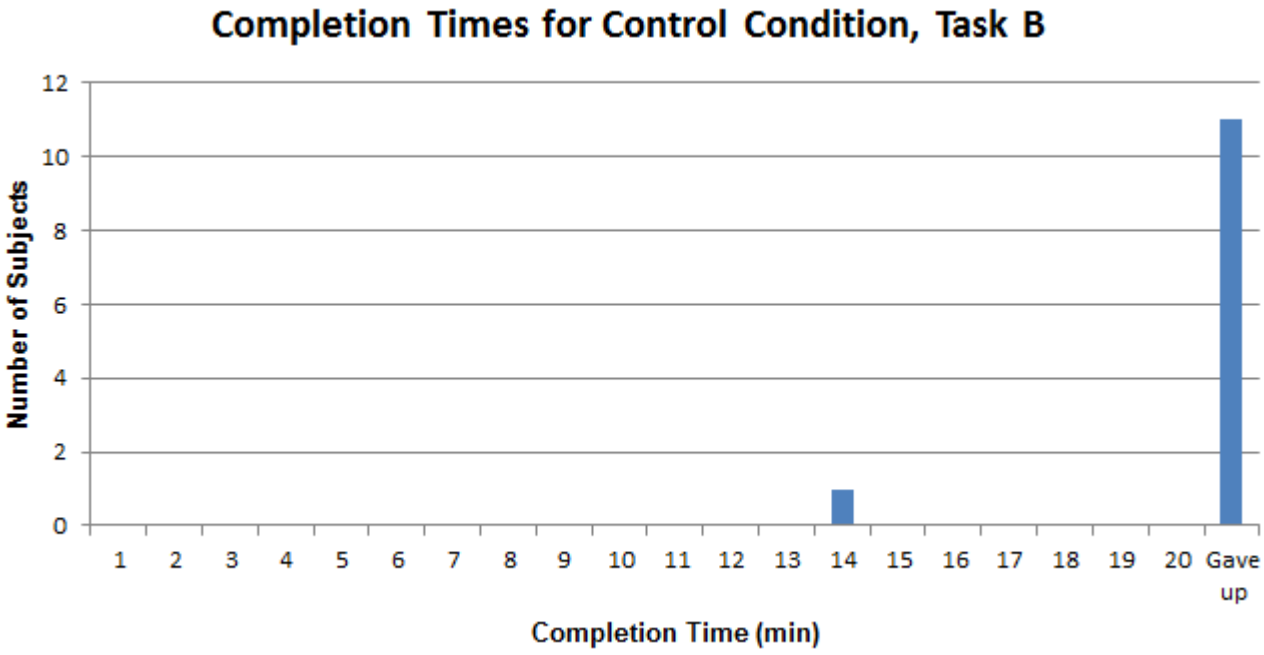

Fig. 17: Our participants' completion times for control condition task B.

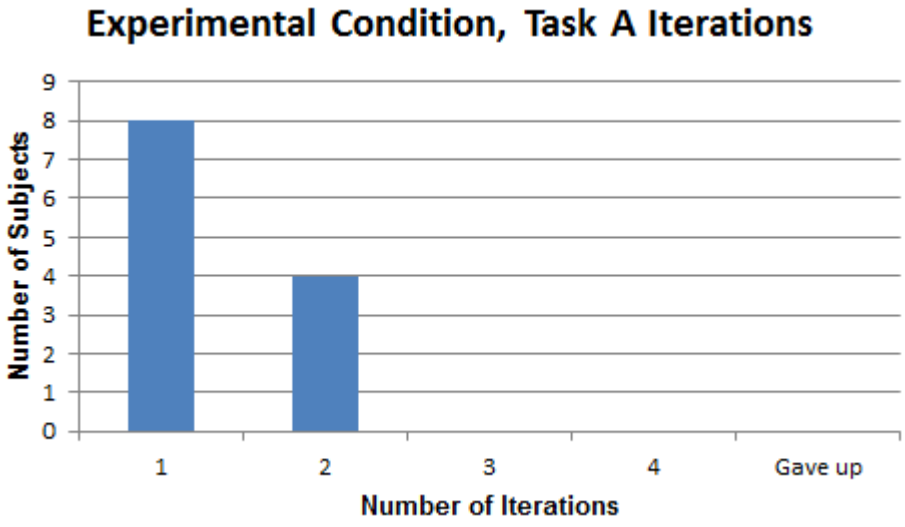

Fig. 18: The number of iterations our participants executed to complete experimental condition task A.

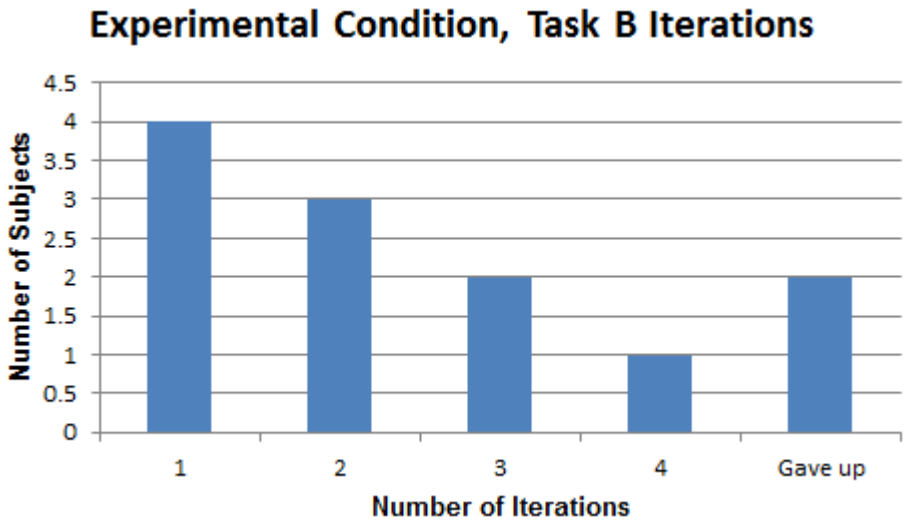

Fig. 19: The number of iterations our participants executed to complete experimental condition task B.

1) Completion Statistics: Our participants finished the experimental condition task A in an average of 2.75 minutes, with a standard deviation of approximately 2. Two participants gave up during experimental condition task B. The rest completed the task in an average of 5.1 minutes, with a standard deviation of 3.4. In comparison, only one person successfully completed control condition task A and B. Everyone else gave up. The one who completed the control conditions task finished task A in 19 minutes, and task B in 14 minutes. These statistics show that our participants overwhelmingly found Quicksilver easier and more efficient compared to the method they came up with in the control condition.

2) Experimental Condition Iteration Statistics: Experimental condition task A took an average of 1.3 iterations to complete, with a standard deviation of 0.49. Experimental condition task B took an average of 2 iterations to complete, with a standard deviation of 1.05. For the subjects who completed these tasks, the number of iterations was well within acceptable limits. The only user to show some frustration at using Quicksilver took the maximum 4 iterations for task B.

3) Control Condition Methods: When asked for the method that they would use in the control condition, eight subjects stated that they would manually copy and paste the data in the correct place (650 rows for task A, 350 for task B). When asked if they could come up with another method, they stated that they could not. One subject used Google to find how to execute a join in Excel. This subject found a post describing how to use `vlookup`, which was exactly what the subject was looking for. However, the subject ignored the writing, stating that "it didn't look like something I'd understand". The last three searched Excel documentations for relevant functions, but only one understood and successfully applied the functions necessary to complete the control condition tasks (`vlookup` for task A, and `isna`, `match` and sort for task B). This successful subject was our most experienced subject, with 10 years of computer science experience. Even with this much experience, the subject encountered many difficulties in completing the task, only finishing task A in 19 minutes and task B in 14 minutes, much slower than the completion times for the experimental condition.

4) Preference: Every subject concluded that they preferred using Quicksilver for table transformations.

5) Learning: Not only do these results indicate that Quicksilver is easy to use, but also that Quicksilver is easy to learn. This is because we only provided the user 5 minutes worth of instruction on the tool. In comparison, Resiner showed that even after 14 hours of academic instruction on SQL, nonprogrammers averaged a 65% on a test asking them to write SQL queries [7].

IX. RELATED WORK

Our paper deals heavily with program synthesis, the creation of programs based on underspecified specifications. Many synthesis techniques that involve demonstrations exist in different fields. In particular, Gulwani has explored

this problem in string transformations [8] and unstructured spreadsheet tables [10].

There has also been work in the database community on generating queries based on user inputs. For example, Zloof created a graphical user interface to database systems that allowed users update the database and query it using examples in Query by Example [5]. His system was implemented in an entirely different fashion and required much more involvement from the user.

Query by Output [6] tries to find a class of queries that, when executed on a given database, produce the same given output. Their goal is to simplify the user's input query and to educate the user on the database schema by providing the user alternative, instance-equivalent queries. Our goal is different from theirs and we synthesize queries based on incomplete outputs, whereas they find projection, selection, and join queries based on complete outputs.

Das Sarma et. al. in Synthesizing Views Definitions From Data [1] attempts to synthesize queries whose outputs are close to a given output (or view) on a single relational table, with no joins, projections, or set operations.

DataPlay [2] is a query tool integrated into databases to simplify query creation for users. It introduces graphical representations of queries that the user can directly manipulate. It also builds new queries based upon user-specified constraints. From these queries, DataPlay shows the user its results, which the user can mark with 'want out' and 'keep in' labels. DataPlay will then find a new query based on these specifications with the smallest change to its last query. Quicksilver differs from DataPlay in several aspects. Quicksilver is designed to operate on various formats of tabular data, not just databases. Quicksilver also has a much simpler demonstrations grammar, whereas DataPlay's graphical query language is somewhat complex, presenting a significant learning curve to the user. Quicksilver's online nature and touch-compatibility allow for the tool to be used in much wider situations as well.

Wrangler [9] is an online data transformation tool. A user can upload a table to this tool, which then suggests transformation operations to the user. The interactions between the user and the tool involve the user selecting areas of the table, and then executing suggested operations from the tool. Wrangler is different from our tool in its language of demonstrations and transformations. We focus on relational transformations, whereas Wrangler focuses on operations such as reshaping tables and splitting data.

Cheung et. al. developed a code analysis algorithm called QBS [3]. QBS transforms fragments of application logic into SQL queries in order to reduce the amount of data sent from databases to applications. It could also help the database optimize these operations. QBS is related to Quicksilver somewhat in its synthesis of queries, however its inputs and applications are vastly different.

X. CONCLUSION

In this paper we presented an online tool that synthesized relational algebra from user demonstrations. We discussed the grammar of demonstrations, the scope of our generated queries, algorithms behind the tool, and limitations that we accepted. There is still some work to be done on this subject, such as finding efficient ways to overcome the limitations of our approach. Another interesting avenue of research would be to find out how to apply these concepts to tables that change frequently in a way that is easy for end-users to understand.

XI. ACKNOWLEDGMENTS

The authors would like to acknowledge the contributions of Bjoern Hartmann.

REFERENCES

[1] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. 2010. Synthesizing view definitions from data. In Proceedings of the 13th International Conference on Database Theory (ICDT '10), Luc Segoufin (Ed.). ACM, New York, NY, USA, 89-103.

[2] Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. 2012. DataPlay: interactive tweaking and example-driven correction of graphical database queries. In Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST '12). ACM, New York, NY, USA, 207-218.

[3] Cheung, Alvin, Armando Solar-Lezama, and Samuel Madden. "Optimizing Database-Backed Applications with Query Synthesis." (2013).

[4] M. Tamer Özsu; Patrick Valduriez (2011). Principles of Distributed Database Systems (3rd ed.). Springer. ISBN 978-1-4419-8833-1.

[5] Moshé M. Zloof. 1975. Query-by-example: the invocation and definition of tables and forms. In Proceedings of the 1st International Conference on Very Large Data Bases (VLDB '75). ACM, New York, NY, USA, 1-24.

[6] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09), Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, 535-548.

[7] Reisner, P., Boyce, R.F., and Chamberlin, D.D. Human factors evaluation of two data base query languages - Square and Sequel. In Proc. Nat. Computer Conf., AFIPS Press, Arlington, Va., 1975, pp. 447-452.

[8] Rishabh Singh, Sumit Gulwani: Learning Semantic String Transformations from Examples. PVLDB 5(8): 740-751 (2012)

[9] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11). ACM, New York, NY, USA, 3363-3372.

[10] W. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation (PEPM '12)*. ACM, New York, NY, USA, 43-52.